A Security Assessment of the Minos Architecture

Jedidiah R. Crandall and Frederic T. Chong
University of California at Davis
Computer Science Department
{crandall, chong}@cs.ucdavis.edu

Abstract

Minos is a microarchitecture that implements Biba's low-water-mark integrity policy on individual words of data. Months of testing have revealed a robust system that stops attacks which corrupt control data to hijack program control flow. The low-water-mark policy is orthogonal to the memory model so that it works with existing software and middleware. The key is that Minos tracks the integrity of all data, but protects control flow by checking this integrity when a program uses the data for control transfer. Existing policies, in contrast, need to differentiate between control and non-control data *a priori*.

Our implementation of Minos for Red Hat Linux 6.2 on a Pentium-based emulator is a usable Linux system on the network. We have demonstrated that Minos protects against a menagerie of real control data attacks, not just buffer overflows. This paper will detail our security assessments of Minos and other hardware and software mechanisms designed to stop the same class of attacks. We conclude that while Minos is substantially more secure than other approaches, existing C programs lack the semantic information necessary to totally secure their control flow. More details about the implementation of Minos are available in [1].

1 Introduction

Control data attacks form the overwhelming majority of remote attacks on the Internet, especially Internet worms, and are a major constituent of local attacks designed to raise privileges. The cost of these attacks to commodity software users every year now totals well into the billions of dollars.

In Minos, every 32-bit word of memory is augmented with a single integrity bit at the physical memory level, and the same for the general purpose registers. This integrity bit is set by the kernel when the kernel writes data into a user process' memory space, with zero meaning low and one meaning high. The kernel can make this decision based on the trust it has for the data being used as control data. *Control data is any function pointer or jump targets such as return pointers on the stack, library pointers in the Global Offset Table, or virtual function pointers in C++ objects.* Minos is very similar to the architecture proposed in [2] but was developed independently, uses a different policy, and the focus is more on the system level.

In Minos Biba's low-water-mark integrity policy [3] is applied by the hardware as the process moves data and uses it for operations. If two data words are added, for example, an AND gate is applied to the integrity bits of the operands to determine the integrity of the result. A data word's integrity is loaded with it into general purpose registers. A hardware exception traps to the kernel whenever low integrity data is used for control flow purposes by an instruction such as a jump, call, or return. Intuitively, any control transfer involving untrusted data is a system vulnerability. Minos detects exactly these vulnerabilities, and consequently avoids false positives under extensive testing. Biba's low-water-mark policy is notorious for its monotonic behavior so we chose to evaluate a full implementation for false positives rather than a handful of statically compiled benchmarks such as SPEC 2000.

This paper is organized as follows. Section 2 is meant to justify the design decisions in terms of Biba's low-water-mark integrity policy. The architectural support required for Minos is described in section 3. Sections 4 and 5 discuss our methodology for evaluating Minos and the security issues that must be addressed at the system level. We developed an exploit to show the insecurity

of the current best practices which is explained in section 6. A security assessment of Minos compared to several other architectures in section 7 is followed by conclusions.

2 Biba's Low-Water-Mark Integrity Policy

This section will explain Biba's low-water-mark integrity policy, why it's an important abstraction for the Minos system, and what exceptions to this policy at the architectural level in Minos may raise security concerns.

Biba's low-water-mark integrity policy [3] specifies that any subject may modify any object if the object's integrity is not greater than that of the subject, but any subject that reads an object has its integrity lowered to the minimum of the object's integrity and its own. The only other implementation of Biba's low-water-mark integrity policy that we know of is LOMAC [4] which applied this policy to file operations and ran into self-revocation problems. This monotonic behavior is the classic sort of problem with the low-water-mark policy, which Minos ameliorates with a careful definition of trust.

An integrity policy was chosen because the confidentiality and availability components of a full security policy are not critical for control data protection. We chose Biba's low-water-mark policy over other integrity policies because it has the property that access controls are based on accesses a subject has made in the past and therefore need not be specified. For a more thorough explanation of this property we refer the reader to [4].

The most obvious tenet of this policy visible in Minos is the AND gate between the integrities of both operands to an operation to determine the integrity of the result. Policy 1 from [2] assumes the result is high integrity even if both operands are low integrity, a necessity if the integrity of addresses used in 32-bit loads and stores is to be checked (this will be explained in section 7). We implemented a similar policy on our Bochs emulator and found that it does not catch this vulnerability:

```
typedef function();
function *f;
scanf("%d", (int *) &f);
f();
```

The *scanf()* function in this case will convert the string read into an integer using an internal version of *strtol()*, which uses computation to do the conversion rather than lookup tables so that it can support a range of bases from 2 to 36.

Basically, this allows arbitrary high integrity values to be placed in memory. These values don't necessarily have to be in a special place to exploit this. For example, if an attacker can get an arbitrary high integrity value anywhere on the stack before exploiting a format string vulnerability they can eat the stack up to this place and write an arbitrary value to an arbitrary location (this arbitrary value will be the sum of size specifiers in the format string which are also converted from a string to an integer).

In Minos there are also information flow problems which may allow an attacker to bypass the low-water-mark policy. Statements like

```
if (LowIntegrityData == 5)
     HighIntegrityData = 5;

HighIntegrityData =
     HighIntegrityLookupTable[LowIntegrityData];

HighIntegrityData = 0;
while (LowIntegrityData--)
     HighIntegrityData++;
```

give an attacker control over the value of high integrity data via information flow.

These were supposed to be pathological cases, but they are not in the case of 8- and 16-bit data because of the way functions like *scanf()* and *sprintf()* handle control characters and also because of translations between strings and integer values or something like the conversion from ASCII to UNICODE exploited by Code Red and Code Red II.

This is why Minos treats 8- and 16-bit data and operations differently from those of 32-bit values. The integrity of the address used in an 8- or 16-bit load or store is checked and the destination word is forced low integrity if the address is low integrity. Without this Minos does not stop Code Red II. Also, 8- and 16-bit immediate values are always assumed to be low integrity. These restrictions prevent attackers from building arbitrary 32-bit values out of smaller data.

Minos also assumes that all misaligned 32-bit reads and writes are low integrity. This way arbitrary 32-bit values cannot be created through "striping" methods.

The Sun Java Just In Time (JIT) compiler produces false positives when 8- and 16-bit immediates are low integrity, but run without any problems when 8- and 16-bit immediates are high integrity. We added a compatibility mode for the JIT, but feel it would be better in terms of

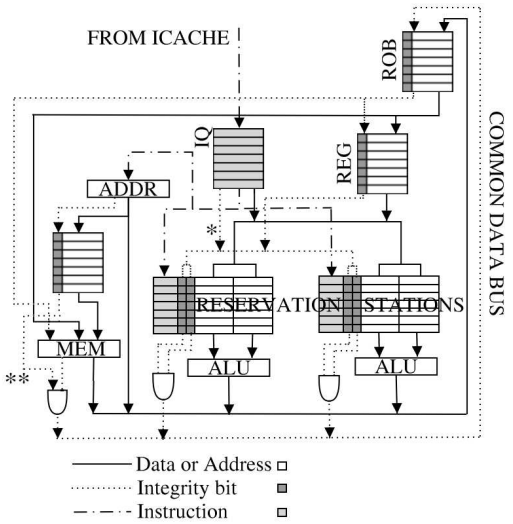


Figure 1: **Minos in an out-of-order execution microprocessor core. *Based on size and compatibility settings. **Ignored for 32-bit loads and stores.**

security if JIT compilers simply used 32-bit immediate values or static variables in place of these 8- and 16-bit immediates that cause the problem.

3 Architecture

The goal of the Minos architecture is to provide system security with negligible performance degradation. To achieve this goal, we describe a microarchitecture which makes small investments in hardware where the tag bits in Minos are in the critical path. Our software Minos emulator only achieves about 10 million instructions per second on a 2.8 GHz Pentium 4.

At a basic level, every 32-bit word of data must be augmented with an integrity bit. This results in a maximum memory overhead of 3.125% (neglecting compression techniques), which can be paid for with Moore's law in 26 days. The real cost, which we will try to address in this section, is the added complexity in the processor core. We argue that this complexity is well justified by the security benefits gained and the high compatibility of Minos with commodity software. Given increasing transistor densities and decreasing performance gains, investments in reliability and security make sense.

Figure 1 shows the basic data flow of the core of a Minos-enabled processor. One bit is added to the common data bus. When data or addresses are transmitted, their integrity bit is also transmitted in parallel. The reorder buffer and the load buffer have an extra bit per tag to store the integrity bit. The reservation stations have two integrity bits, one for each operand. The integrity of the result is determined by applying an AND gate to the integrity bits of the operands. All of the integrity bit operations can be done in parallel with normal operations and are never in the critical path, and there is no need for new speculation mechanisms.

The L1 cache in a modern microprocessor, the Pentium 4 for example, is typically about 8KB and is optimized for access time. To maintain this low access time, we store the integrity bit with every 32-bit word as a 33rd bit. The total storage overhead in an L1 cache of this size is 256 bytes. The on-chip L2 cache, on the other hand, can be as large as 1MB and is optimized for hit rate and bandwidth. To keep the area overhead low and the layout simple, we use the same technique often used for parity bits: have one byte of integrity for every 256 bit cache line.

All of the floating point, MMX, BCD, and similar extensions can ignore the integrity bits and always write back to memory with low integrity. This is because control data, such as jump pointers and function pointers, are never calculated with BCD or floating point. One possible exception is that MMX is sometimes used for fast memory copies, so these instructions should just preserve the integrity bits. The instruction cache, trace cache, and branch target buffer must check the integrity bits with their inputs, but do not need to store the integrity bits after the check. If data is low integrity, it is simply not allowed into the instruction cache or branch target buffer. Overall, the L1 cache and processor core's area increases will be negligible compared to the L2 cache, so we can produce an estimate of the increase in die area for Minos by looking at the L2 cache alone.

Intel's 90 nm process can store 52 Mbits, or 6.5 MB, in 109.8 mm^2 with 330 million transistors [5]. A 1 MB L2 cache without the extra integrity bits in this process would be about 51 million transistors and 16.9 mm^2. Minos would add to this another 1.59 million transistors and 0.53 mm^2 for an additional 32 KB. The Prescott die area is reported to be 112 mm^2, so the contribution of the extra storage required by Minos in the L2 cache to the entire die area is less than one half of one percent. Using the die cost model from [6] and assuming 300mm wafers, α = 4.0, and 1 defect per cm^2 this is less than a penny on the dollar.

A 32-bit microprocessor without special addressing

modes can address 4 GB of DRAM off chip. This requires 128 MB to store the integrity bits outside the microprocessor. We propose a separate DRAM chip which we will call the Integrity Bit Stuffer (IBS). The IBS can coexist with the bus controller and store the integrity information for data in the DRAM. When the DRAM fills requests for data, the IBS stuffs the stored integrity bits with this data on the bus.

By using a banking strategy that mirrors that of the conventional DRAM chip it can be guaranteed that the integrity bit will always be ready at the same time as the conventional data. The bus must be widened from 64 to 66 bits. When the data bus is driven by other devices for DMA or port I/O, the IBS assumes high integrity.

The hardware support needed for Minos is almost identical to what is needed for the soft error rate reduction mechanism proposed in [7]. The same paper discusses other uses of tag bits. The PowerPC AS has a tag bit per 64-bits and is used for running the microcode of iSeries programs. A 64-bit Linux implementation with Minos support on the iSeries may be possible by using a similar microcode approach.

4 Methodology

We emulated Minos on a Pentium emulator called Bochs [8]. Bochs emulates the full system including booting from the BIOS and loading the kernel from the hard drive. DMA, port I/O, and extensions like floating point, MMX, BCD and SSE are supported. The floating point and BCD instructions ignore the integrity of their inputs and their outputs are always low integrity. A single integrity bit was added to every 32-bit word in the physical memory space. All port I/O and DMA is assumed to be high integrity so that all existing devices and drivers are compatible with Minos.

The Pentium is also byte and 16-bit word addressable but it suffices to only store one integrity bit for every 32-bit word. Compilers align all control data along 32-bit word boundaries for performance reasons [9]. If a low integrity byte is written into a high integrity 32-bit word, or a high integrity byte is written into a low integrity word, the entire resulting word is then low integrity. The same applies to 16-bit manipulation of data. More restrictions on the manipulation of 8- and 16-bit data was explained in section 2.

Every instruction operation applies the low-water-mark integrity policy to its inputs to determine the integrity of the result. All 8-bit and 16-bit immediate loads are low integrity unless the processor is running in a special compatibility mode, and all memory references to load 8-bit and 16-bit values also have the low-water-mark integrity policy applied to the addresses used for the load, something that is needed for our Windows implementation of Minos to catch Code Red II [1] as has already been stated.

String operations on the Pentium, such as a memory copy, go from one segment to another. To give the kernel the ability to mark data low integrity as it copies it into a process' memory space the reserved 53rd bit in a Pentium segment descriptor entry is interpreted to mean that data written into this segment should be forced low integrity. If the 53rd bit of the segment descriptor is not set then the integrity bit is simply copied.

Performance results of Minos' implementation of virtual memory swapping, a discussion of false positives, and tests of Minos with existing exploits are available in [1].

5 Operating System Changes

Details about the Minos-specific changes we made to the Linux kernel are available in [1]. This section will address some system-level security concerns.

At a system level Minos basically defines trust based on how long data has been part of the system. On a *read()* system call the *ctime* and *mtime* of the inode are checked and any data created or modified after an *establishment time* is forced low integrity as it is written into a process' address space. An exception was made for pipes between lightweight processes that share the same address space for compatibility with pthreads. It is not productive toward an attack for one thread to hijack the control flow of another thread if they share the same address space and kernel data structures.

Minos secures programs against attacks that hijack their low-level control flow by overwriting control data. The definition of trust in our Linux implementation stops all remote intrusions based on control data corruption. We protect against local control data attacks designed to raise privileges but only because the line between these and remote vulnerabilities is not clear.

Virtually all remote intrusions are control data attacks. The exceptions are directory traversal in URLs (for ex-

ample, "http://www.x.com/../../system/cmd.exe?/cmd"), control characters in inputs to scripts that cause the inputs to be interpreted as scripts themselves, or unchanged default passwords. These kinds of software indiscretions are outside the scope of what the architecture is responsible for protecting.

Code injection attacks are a subset of control data attacks where arbitrary machine code is executed. Hijacking the control flow of a program, with or without injecting arbitrary machine code, is a very powerful tool for an attacker. Minos was not designed to protect important data other than control data such as file descriptors or pointers to filenames.

For compatibility with existing hardware without the need for new device drivers all DMA and port I/O is considered high integrity until the kernel forces the data low integrity during a *read()* system call. The read() system call is used for reading from files, network sockets, pipes, and just about everything else in Linux. The only other way for data to enter a process' address space is through other system calls but typically a remote attacker's inputs will be introduced through a network socket. This network socket read will force the data low integrity and if Biba's low-water-mark policy is enforced properly it will never go up in integrity. Any object in the virtual file system that it is written to will have its *ctime* changed to the current time and won't pass the establishment time requirement, so data can't be lifted to high integrity by going out to the disk and coming back, for example.

The alternative to assuming all DMA and port I/O is high integrity would be to force it low integrity and have a segment descriptor that lifts data up to high integrity on certain conditions. This would be a violation of the low-water-mark policy. Naturally, the next issue to address is the restoration of high and low integrity marks during virtual memory swapping.

There is another special segment descriptor in Minos which, when used in string operations, causes the source or destination to have a stride of 32 words and the value copied in or out of this segment is the 32 bits of integrity information for this 32 word block. This way the kernel can copy the integrity information from an entire 4 KB page into a 128 byte buffer, or copy the integrity information of a 128 byte buffer into the integrity bits of an entire page to enable virtual memory swapping.

An obvious security concern is that this may violate the low-water-mark policy if an operation is available to make data be lifted to high integrity, so this special segment descriptor only changes high integrity marks to low integrity, but never vice-versa. This preserves the low-water-mark policy and does not create any false positives because after a page is read back from a swap device it will be all high integrity until the kernel tries to map it at which point the appropriate low integrity marks are restored.

Minos also checks the integrity of all *mmap()ed* files such as dynamically linked libraries. However, the originally mounted binary is not checked for the establishment time requirement so that programs can be compiled statically and run without changing the establishment time. If a compiled static binary is flushed to disk with *sync()* it can be run without causing a Minos alert. To exploit this an attacker would need to already have a shell or the equivalent, and Minos is designed to stop remote attackers before they can obtain a shell. We make no claim to security against local privilege escalation.

Another concern is that Minos places a lot of trust in the operating system but system code sometimes has the same bugs as application code. For example, the Linux kernel version we used is vulnerable to an integer overflow in the *do_brk()* function [10]. It requires that the attacker already has a shell because they need to mount an ELF binary with no stack. It basically allows the kernel's pages above the 0xc0000000 address to be mapped into the process' page table. These pages are at level 0 (system) and the process is in level 3 (user space) so the pages are still protected by the absence of read and write permission bits for level 3.

These protections are meaningless in Linux once a process has a page in its address space though, so in practical terms the process can arbitrarily read from and write to kernel space. Writes are performed by writing the data to a file and then reading the file using a pointer to a buffer that is in the kernel space. Reads are performed by doing the inverse. We believe that execute permission bits on pages will prove to be equally as meaningless for pages in a process' address space.

We know of no remotely exploitable control data vulnerabilities in the Linux kernel where the attacker would not already need to have a shell. Memory corruption vulnerabilities in the kernel are not limited to control data attacks because the kernel stores a lot of important system information. Thus Minos checks the integrity of control flow transfers in system mode but no claim is made of protecting the kernel from users with a shell.

6 The Hannibal Exploit

We developed the *hannibal* exploit to illustarate the insecurity of current best practices. This section will describe our estimation of current best practices, including non-executable pages, return pointer protection, random library placement, and some specifics of Windows XP Service Pack 2, and then will describe an exploit not addressed by any of these techniques for an old vulnerability in the *wu-ftpd* service of Linux.

Non-executable pages are already available on 64-bit Pentium-based architectures and both Windows XP Service Pack 2 and Linux kernel 2.6 support their use even in 32-bit mode. Attacks are well known to subvert these protections [11] but require that multiple functions be linked together using return addresses as function pointers and building a chain of stack frames. This is easily averted by return pointer protection such as StackGuard [12] or that available with Windows compilers. It is also common to place libraries at random locations as is possible with Gentoo Linux, or place them in parts of memory with zeroes in the address which is the default for Fedora Core 2.

In addition to Hardware Data Execution Prevention (DEP) (the name for non-executable pages support in Windows XP Service Pack 2), Software DEP protects Windows-critical processes by default. For programs compiled with Secure Structured Exception Handling (SEH) pointers to exception handlers are only followed if they match the pointer of a properly registered exception handler function. For programs without Secure SEH the function pointer is only checked to make sure it addresses a portion of memory marked as executable (regardless of hardware support). Code Red II overwrote one of these SEH function pointers with a pointer to code in an executable portion of memory containing "CALL EBX" to jump back to the stack, and therefore would not have been stopped by software DEP. Furthermore, even with Secure SEH critical flaws were pointed out in [13] and it's not clear if Windows XP Service Pack 2 addressed these concerns or not. All of this casts doubt on whether Service Pack 2 can stop the control data attacks of the past let alone those of the future.

To stop control data attacks we must protect the integrity of all control data and stop the attack before control flow is hijacked. To further illustrate this point we assumed non-executable pages, return pointer protection, and random placement of library functions on our Red Hat Linux 6.2 Bochs emulator with Minos disabled and were easily able to still obtain a remote root shell. With Minos enabled this attack is stopped at the first illegitimate control flow transfer.

The *hannibal* exploit takes advantage of the use of a Procedure Linkage Table (PLT) and Global Offset Table (GOT) to facilitate calls to dynamically linked functions from statically compiled code. The following C program is complex enough to require the use of a PLT and GOT:

```
#include <stdio.h>

int main()
{
     printf("Hello World!\n");
     return 0;
}
```

The main program is compiled with the value 0x08048268 statically bound to *printf()*. This three instruction sequence is the PLT entry for *printf()* and resides in read-only, executable memory:

```
0x8048268 <printf>:
     jmp    *0x80494b8
0x0804826e <printf+6>:
     push   $0x8
     jmp    0x8048248 <_dl_runtime_resolve>
```

The GOT entry for *printf()* is loaded from 0x080494b8 (readable and writable memory) and an unconditional jump either reads the value 0x0804826e which will continue to push an identifier for *printf()* and jump to a function to resolve the symbol and update *printf()'s* GOT entry, or will jump directly to *printf()* if the symbol has already been resolved.

Figure 2 outlines the steps of the *hannibal* exploit. The *wu-ftpd 2.6.0* FTP server daemon for Red Hat 6.2 contains a format string vulnerability which allows us to write an arbitrary value into a nearly arbitrary location in memory without touching the stack or crashing the process [14]. In short, the *hannibal* exploit uploads a statically compiled binary executable called "jailbreak" via anonymous FTP onto the victim machine and replaces *rename(char *, char *)'s* GOT entry with a pointer to *execv(char *, char **)*'s PLT entry. Subsequently a request to rename the file "jailbreak" to "\xb8\x6b\x08\x08" will cause the server to run *execv("jailbreak", {"jailbreak", NULL})*.

As a practical matter the string "\xb8\x6b\x08\x08" must land on the heap in a chunk initially with all zeroes in it because *execv()* expects a NULL-terminated list of

arguments. This is achieved by changing *syslog(int, char *, int)'s* GOT entry to point to the PLT entry for *malloc(int)* and trying to login sixty times which will generate system log events because we're already logged in. This memory leak will "squeeze the heap" the way Hannibal squeezed the Roman infantry at the Battle of Cannae and cause our string to land in the wilderness chunk.

The "jailbreak" executable will inherit the network socket descriptors of the *wu-ftpd* daemon, break out of the *chroot()* jail keeping it in "/home/ftp" using well-known techniques, and execute a root shell. A couple of interesting points can be made about this exploit. The first is that the *execv()* symbol is not even resolved until the attack hijacks control flow and jumps to *execv()'s* PLT entry which will locate this function and resolve the symbol for us. Also, most format string vulnerabilities, including the one used here, make it trivial to produce either an arbitrary write primitive or an arbitrary read primitive [15]. Randomizing the locations of the PLT, GOT, or even the static binary won't help at all because the attacker can easily use arbitrary read primitives to locate them. Sandboxes don't help either because an arbitrary write primitive allows the attacker to simply bypass the sandbox and jump to code directly after the point where the checks are passed.

1. Log into the FTP server as anonymous
2. Upload the jailbreak executable binary via anonymous FTP
3. Use format string arbitrary write primitive to overwrite *rename(char *, char *)'s* GOT entry with a pointer to the PLT entry of *execv(char *, char **)*
4. Use format string arbitrary write primitive to overwrite *syslog(int, char *, int)'s* GOT entry with a pointer to the PLT entry of *malloc(int)*
5. Try to log in sixty more times generating sixty *syslog()* events which in actuality become sixty small *malloc()s* to "squeeze the heap"
6. Use "RNFR jailbreak" and "RNTO \xb8\x6b\x80\x08" to request that the server execute *rename("jailbreak", "\xb8\x6b\x80\x08")* which if "\xb8\x6b\x80\x08" lands in the wilderness chunk is actually *execv("jailbreak", {"jailbreak", NULL})*
7. The *_dl_runtime_resolve()* function will resolve the symbol for the *execv()* function, locate it, put the appropriate pointer in *execv()'s* GOT entry and continue the call
8. jailbreak inherits the network socket descriptors on stdin and stdout and does the following:

   ```
   seteuid(0);
   setegid(0);
   mkdir("/tmp", 777);
   chroot("/tmp");
   chdir("../..");
   chroot(".");
   execv("/bin/bash", {"/bin/bash", NULL});
   ```

   This is necessary because the ftp daemon is kept in a chroot("/home/ftp") jail so that "/" is "/home/ftp" until we break out of it.

Figure 2: **Steps in the Hannibal Exploit**

7 Security Assessment

We have demonstrated that Minos stops all sorts of existing control data attacks [1], but we must address the security of Minos against future kinds of attacks developed with subversion of Minos in mind. A useful way to think of how attacks more advanced than simple buffer overflows are developed is to consider that vulnerabilities lead to corruption, corruptions lead to primitives (such as an arbitrary write), and primitives can be used for higher level attack techniques [16].

We will compare the security of Minos specifically to the AS/400 [17], the Elbrus E2K [18], a similar architecture with a different policy [2], and the current best practices. Our estimation of the current best practices is execute permissions on pages, random placement of library routines in memory, and return pointer protection such as StackGuard [12].

The following three classes of control data attacks must be considered: 1) Can an attacker overwrite control data with untrusted data undetected? 2) Can an attacker

cause the program to load/store control data to/from the wrong place? and 3) Can an attacker cause the program to load control data from the right place but at the wrong time?

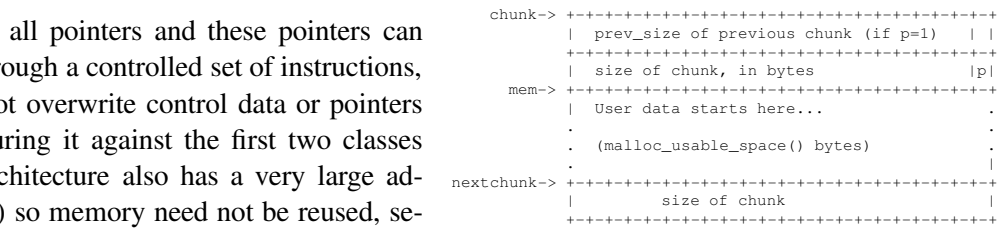
The AS/400 tags all pointers and these pointers can only be modified through a controlled set of instructions, so an attacker cannot overwrite control data or pointers to control data securing it against the first two classes of attacks. This architecture also has a very large address space (64-bits) so memory need not be reused, securing it against the third class of attacks which have a temporal element. The AS/400 is secure against control data attacks when this pointer protection is enabled, but these protections are disabled for Linux on the iSeries [19] simply because C programs written for Linux don't have the semantic information to distinguish pointers from other data.

The Elbrus E2K uses strong runtime type-checking to protect the integrity of all pointers, and pointers may not be coerced with other data types such as integers. To protect itself against temporal reference problems C/C++ programs may not have unchecked references from data structures with a longer lifetime to those with a shorter lifetime (such as from the stack to the heap) and C++ programs may not redefine the *new* operator. These constraints are very draconian but would be necessary to totally secure C/C++ programs against all three classes of control data attacks.

Section 6 already discussed current best practices and how easily they are subverted. It is necessary to protect the integrity of all control data and without hardware support it is really only feasible to protect the integrity of return pointers on the stack and do a few simple checks. But any control data left anywhere in memory unprotected can be the victim of an arbitrary write primitive.

Minos stops this kind of attack because Minos protects the integrity of all control data, not just return pointers on the stack. The possible security problems we foresee for Minos are copying valid control data over other control data (which falls in the second class), dangling pointers to control data (which falls in the third class), and generating arbitrary high integrity values through legitimate control flow (which falls in the first class).

Minos prevents all attacks that overwrite control data with untrusted data. To stop attacks that copy other high integrity data over control data Minos would need to check the integrity of addresses used for 32-bit loads and stores, as is done in the both policies of [2]. To see why this is infeasible consider this example of how Doug Lea's *malloc* (which is used in glibc) stores management information on the heap and uses it to calculate pointers:



The *size* field is always divisible by eight so the last bit (*p*) is free to store whether or not the previous chunk is in use. The addresses of all chunks are calculated using the *size* and *pre_size* integers (note that this is a violation of the Elbrus E2K's constraint that pointers may not be coerced with integers). These sizes may be read directly from user input so you would expect them to be low integrity. That means that all heap pointers will be low integrity if the integrity of these sizes is checked, and if it is not checked then an attacker can use this fact to modify heap pointers undetected. These sizes are never bounds-checked because they are supposed to be consistent with the size of the chunk.

If all heap pointers are low integrity then all control data or pointers to control data on the heap will also become low integrity when they are loaded or stored using these pointers. An example of control data or pointers to control data on the heap might be C++ virtual function pointers or plug-in hooks. This will create a lot of false positives. That is why both the integrity of addresses used for loads and stores of control data and the integrity of all operands to an operation cannot be checked without producing false positives. Thus the first policy of [2] is able to do the former and Minos is able to do the latter but neither is able to do both.

The second policy in [2] attempts to do both by assuming that all low-integrity values that are used in a compare operation or a logical AND/OR are bounds checked and therefore safe to be lifted to high integrity. The bit *p* from the *malloc* header above is extracted with a logical AND from the *size* field but this is not a bounds check so an attacker could write an arbitrary even integer into the *size* field and it would become high integrity.

Attackers often use these *size* and *prev_size* fields for heap corruption attacks. If these fields remain high integrity (because of the logical AND) to keep all heap pointers high integrity and avoid false positives then the

following macro may be run during a double *free()* or heap buffer overflow exploit with *P*, *P->fd*, and *P->bk* all high integrity and *P->fd* and *P->bk* may be arbitrary (*BK* and *FD* are temporary values):

```
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  FD->bk = BK;
  BK->fd = FD;
}
```

This acts like an arbitrary write primitive but really what it is doing is making two pointers in memory point to the locations of one another. Thus you can't change a pointer to point to code in non-writable memory and it's not clear if this is exploitable with [2], especially with non-executable pages. From our experience, however, it is never safe to assume that a vulnerability is not exploitable without something explicit in the security mechanism to stop it.

Arbitrary copy primitives appear to be much harder to achieve than arbitrary write primitives. One possibility would be to overwrite both the source and destination pointers of a *memcpy(void *, void *, size_t)*, but both arguments would have to be in writable memory. The *strcpy(char *, char*)* function manipulates data at the byte level so the integrity of the addresses is checked.

We don't believe arbitrary copy attacks will be a problem, but if they are we propose a Sandboxed PLT (SPLT), which splits pointers to critical library functions (like *execv()*, *system()*, or *chroot()*) that aren't performance critical in the GOT into two pieces with an XOR using a 32-bit hash value of the library's symbol. Then the attacker would need not just an arbitrary copy primitive but an arbitrary copy and XOR at the same time. Calls to the SPLT would run special sandboxing code to check their validity, as would the library functions to check that the call came from the SPLT. These sandboxes cannot be subverted with an arbitrary write primitive in Minos, and arbitrary copy attacks would need to be lucky enough to find somewhere in memory the exact high-integrity value needed to bypass the sandboxes.

We don't believe that dangling pointers are practical to exploit in Minos either, because the attacker can't put arbitrary data into the location where the valid control data is expected, it would have to be high-integrity data, so in practical terms they would need an arbitrary copy primitive.

Note that an arbitrary read primitive and an arbitrary write primitive (both of which are trivial with, for example, a format string vulnerability) don't give the attacker an arbitrary copy primitive in Minos because anything which goes through the filesystem and comes back will be low integrity.

One method of generating high integrity arbitrary values might be to exploit a format string vulnerability but use "%s" format specifiers instead of "%9999u", where "%s" is supplied a pointer to a string that is 9999 characters long (a controlled increment). Fortunately, this arbitrary value will be low integrity in our Minos implementation because the count of characters is kept by adding 8-bit immediates to an initially zero integer and our policy treats all 8-bit and 16-bit immediates as low integrity. The first policy in [2] will stop such an attack because the integrity of all 32-bit loads and stores is checked, unless an arbitrary high integrity value can be placed on the stack using the method described in section 2.

8 Related Works

We have cited several related architectures and relevant references throughout this paper, but for a more complete list of related works we refer the reader to [1].

9 Conclusions

We can't say peremptorily that Minos is totally secure against control data attacks for every possible program but we will assert that it is very "securable." As an analogy, consider that the AS/400 is possibly the most "securable" architecture in the world against unauthorized remote access to files but without special procedures to properly secure it remote attacks can be trivial. For example, an attacker might obtain access to an account because of an unchanged default password [20] and then due to a vulnerability such as [21] be able to execute arbitrary commands.

Slight modifications to the library mechanisms and sandboxes in key areas, such as the SPLT, could secure a Minos system against remote control data attacks with a high degree of assurance by taking away primitives like arbitrary copies or controlled increments, and would constitute code changes in centralized locations but not a change to the memory model expected by applications.

10 Acknowledgments

This work was supported by NSF ITR grant CCR-0113418, an NSF CAREER award and U.C. Davis Chancellor's fellowship to Fred Chong, and a United States Department of Education GAANN grant #P200A010306 as well as a 2004 Summer Research Assistantship Award from the U.C. Davis GSA for Jed Crandall. We would like to thank many people who discussed this project with us or read earlier versions of the paper, including Diana Franklin, Mark Oskin, John Oliver, S. Felix Wu, Matt Bishop, and anonymous reviewers. We would also like to thank the developers of the Bochs project.

References

[1] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *The 37th International Symposium on Microarchitecture*, 2004.

[2] G. E. Suh, J. W. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[3] K. J. Biba, "Integrity Considerations for Secure Computer Systems," in *MITRE Technical Report TR-3153*, Apr 1977.

[4] T. Fraser, "LOMAC: Low Water-Mark Integrity Protection for COTS Environments," 2000.

[5] Intel, "Press Release, 12 March 2002."

[6] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach 3rd. ed.* San Mateo: Morgan Kaufmann, 2003.

[7] C. Weaver, J. Emer, and S. S. Mukherjee, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proceedings of the 31st annual International Symposium on Computer Architecture*, p. 264, IEEE Computer Society, 2004.

[8] "Bochs: the Open Source IA-32 Emulation Project (Home Page), http://bochs.sourceforge.net."

[9] "Intel Optimization Manual Order Number 242816-003, Section 3.4."

[10] CERT, "Vulnerability Note VU 301156."

[11] Nergal, "The advanced return-into-lib(c) exploits: PaX case study, Phrack 58."

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. of the 7th Usenix Security Symposium*, pp. 63–78, Jan 1998.

[13] D. Litchfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server (http://www.packetstormsecurity.com/papers/bypass/defeating-w2k3-stack-protection.pdf)."

[14] "Security Focus Vulnerability Notes, bugtraq id 1387."

[15] scut, "Exploiting Format String Vulnerabilities."

[16] jp, "Advanced Doug lea's malloc() exploits, Phrack 61."

[17] National Security Agency, "Final Evaluation Report, International Business Machines Corporation Application System 400."

[18] B. Babayan, "Security, www.elbrus.ru/mcst/eng/ SECURE_INFORMATION_SYSTEM_V5_2e.pdf."

[19] D. Boutcher, "The Linux Kernel on iSeries."

[20] mechanic, "The basics and an introduction to the IBM AS/400, www.9x.tc."

[21] securitytracker.com, "SecurityTracker Alert ID: 1005891."